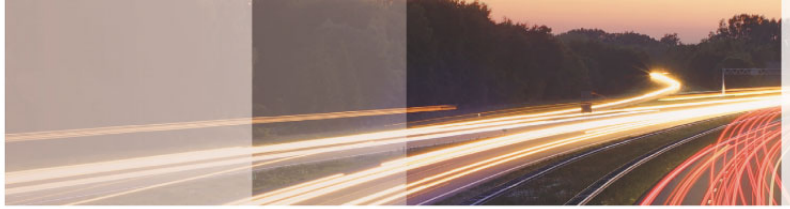


GARR

The Italian Academic & Research Network

www.garr.it

La programmazione di rete IPv6

Come scrivere applicazioni IPv6 compliant in C/C++, JAVA, Perl e Python

v1.1

Mario Reale, Rino Nucara GARR

GARR WS9 – Roma, 15-18 Giugno 2009

Consortium GARR

Contenuti

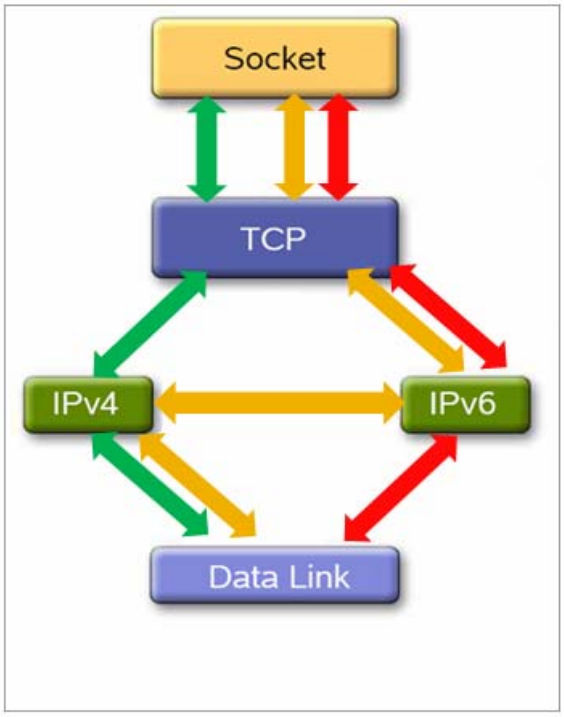
- Concetti generali sulla programmazione AF-independent
- Introduzione alla programmazione IPv6 in
  - C
  - Perl
  - Python
  - Java
- Librerie di Alto Livello (High Level Networking Libraries)

Programmazione AF-independent: l'idea

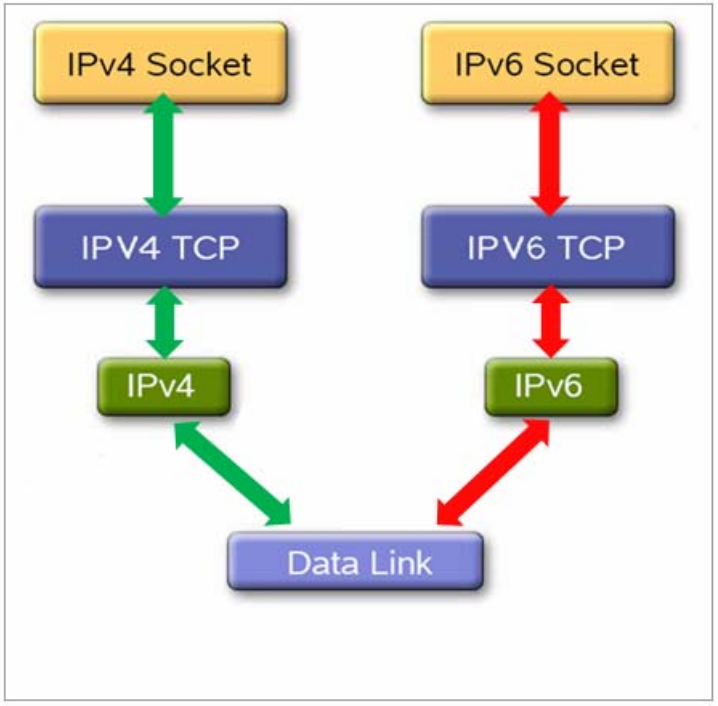
- IPv4 ed IPv6 coesisteranno per svariati anni
- Servers e Clients IPv**X** dovranno connettersi a Clients e Servers IPv**X**
- Per scrivere applicazioni compatibili sia con IPv4 che con IPv6 ci sono due possibilita':
  - 1) Usare **un solo socket IPv6**, in grado di gestire entrambe IPv4 ed IPv6
    - IPv4 puo' essere vista come un caso speciale di IPv6 *(IPv4-mapped addresses)*
  - 2) Aprire **un Socket per IPv4** e **un Socket per IPv6**.
    - Impendendo ai Socket IPv6 di gestire le connessioni IPv4
- Per creare applicazioni AF-independent bisogna non assumere a-priori una versione specifica del protocollo IP
  - A questo scopo sono state introdotte nuove API e strutture dati

Dual stack e separated stack

Sistema Operativo
DUAL STACK



Sistema Operativo a STACK SEPARATI
(obsoleto)



IPv4 traffic

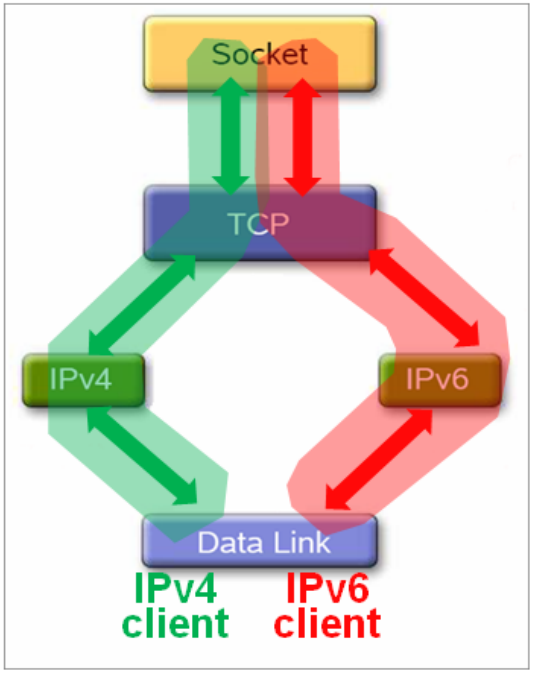
IPv4 mapped into IPv6 traffic

IPv6 traffic

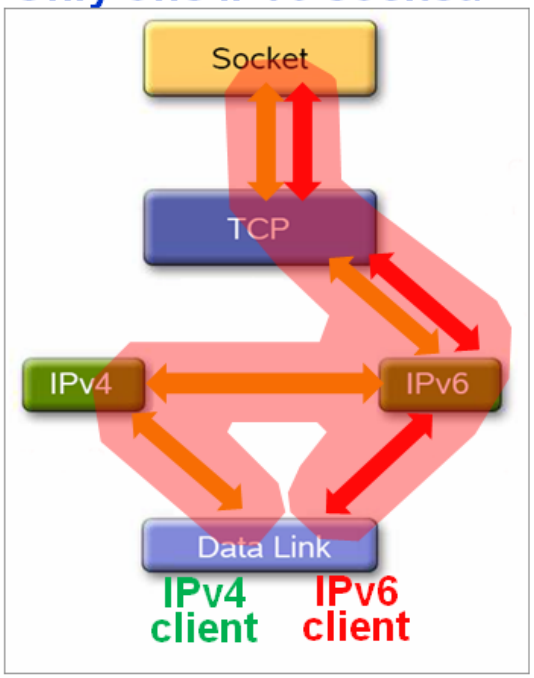
Due tipi di socket server

Nei sistemi Dual Stack, per poter accettare connessioni sia IPv4 che IPv6, i socket servers possono essere costruiti in due modi:

Two sockets:



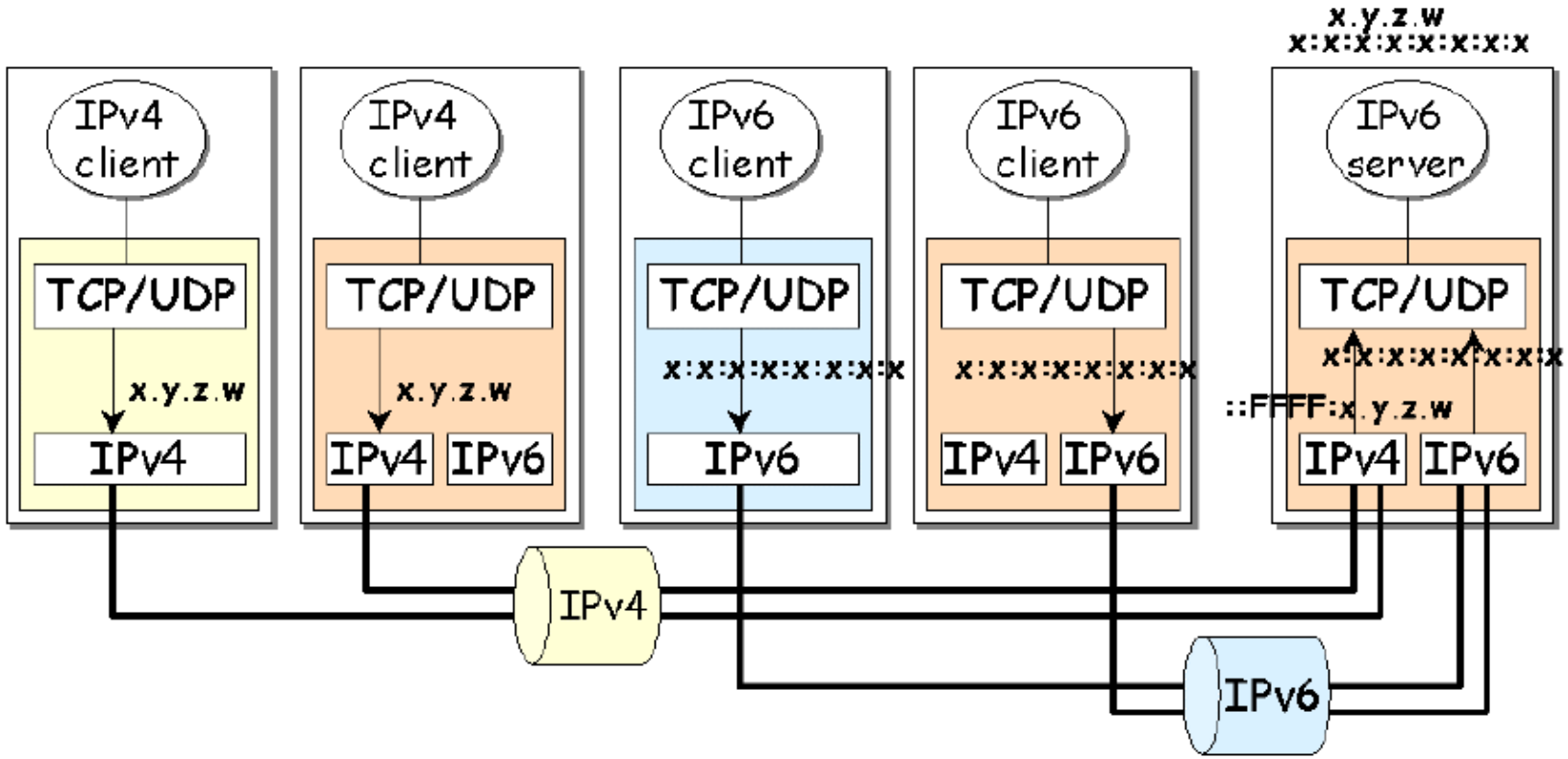
Only one IPv6 socket:



Traffico collegato ad un socket IPv4

Traffico collegato ad un socket IPv6

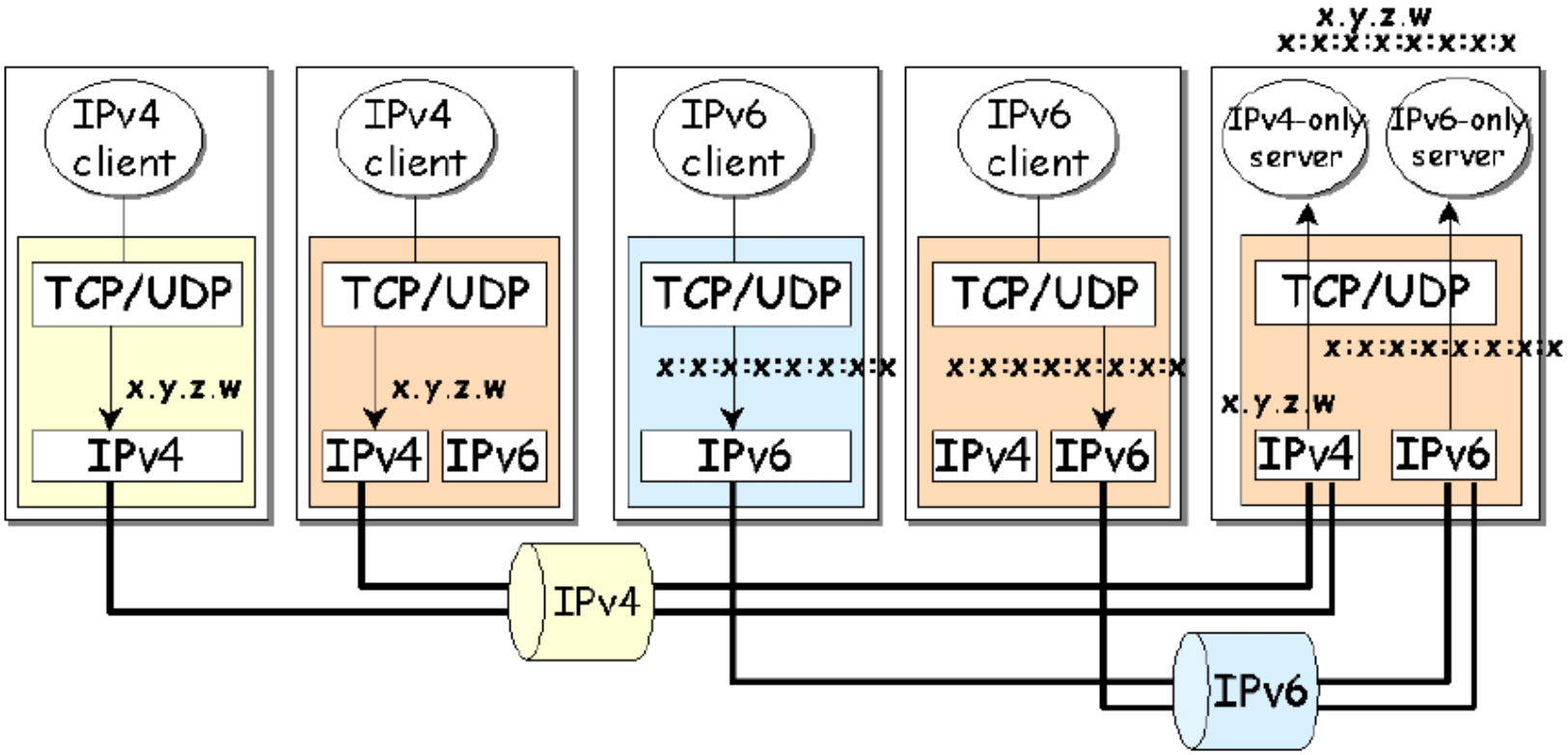
Un solo socket



Dual Stack

Single IPv4 or IPv6 stacks

Due socket



Dual Stack or separated stack

Single IPv4 or IPv6 stacks

Introduzione alla programmazione IPv6 in C

API IPv6

- L' **IETF** prevede due gruppi di estensioni:**RFC 3493** & **RFC 3542**.
- **RFC 3493 Basic Socket Interface Extensions for IPv6**
- Il piu' recente (il successore di RFC 2133 ed RFC 2553. Soprannominato "2553bis")
  - Fornisce definizioni standard per
    - Funzioni Socket di core
    - Address data structures (Strutture dati per gli indirizzi)
    - Funzioni di traduzione Name-to-Address
    - Funzioni per la conversione degli indirizzi
- **RFC 3542 Advanced Sockets Application Program Interface (API)**
  - E' il piu' recente ed e' il successore di RFC2292 (detto anche "2292bis")
  - Definisce interfacce per accedere ad informazione speciale IPv6:
    - IPv6 header
    - Extension Headers
    - Estendere le possibilita' dei raw sockets IPv6

Un nuovo nome per l' address family

- Un nuovo *address family name*, **AF_INET6,** e' stato definito per IPv6
- Corrispondentemente la *protocol family* e' **PF_INET6**

```
#define   AF_INET6   10
#define   PF_INET6   AF_INET6
```

Codice IPv4 :

```
socket(PF_INET,SOCK_STREAM,0); /* TCP socket */
socket(PF_INET,SOCK_DGRAM,0); /* UDP socket */
```

Codice IPv6 :

```
socket(PF_INET6,SOCK_STREAM,0); /* TCP socket */
socket(PF_INET6,SOCK_DGRAM,0); /* UDP socket */
```

Strutture Address Data

- IPv4
  - *struct* ***sockaddr_in***
  - *struct* ***sockaddr***
- IPv6
  - *struct* ***sockaddr_in6***
- **IPv4, IPv6, ...**
  - *struct* ***sockaddr_storage***

IPv4 Address Data Structures


```
struct sockaddr {
    sa_family_t   sa_family;    // address family, AF_xxx
    char          sa_data[14];  // 14 bytes of protocol address
};
```

```
struct in_addr {
    uint32_t s_addr; // 32-bit IPv4 address (4 bytes)
                     // network byte ordered
};

struct sockaddr_in {
    sa_family_t    sin_family;  // Address family (2 bytes)
    in_port_t      sin_port;    // Port number (2 bytes)
    struct in_addr sin_addr;    // Internet address (4 bytes)
    char           sin_zero[8]; // Empty (for padding) (8 bytes)
}
```

IPv6 Address Data Structures (1/3)


```
struct in6_addr {
          uint8_t  s6_addr[16];  // 128-bit IPv6 address (N.B.O.)
};
struct sockaddr_in6 {
  sa_family_t     sin6_family;   //AF_INET6
  in_port_t       sin6_port;     //transport layer port # (N.B.O.)
  uint32_t        sin6_flowinfo; //IPv6 flow information (N.B.O.)
  struct in6_addr sin6_addr;    // IPv6 address
  uint32_t        sin6_scope_id; //set of interfaces for a scope
}
```

- **sockaddr_in6** contiene indirizzi IPv6 ed e' definita una volta incluso l'header <**netinet/in.h**>
- **sin6_family** sovrascrive il campo **sa_family** quando il buffer e' castato a una **sockaddr** data structure. Il valore di questo campo deve essere AF_INET6. (2Byte)
- **sin6_port** contiene il numero di porta UDP o TCP (16bit). Questo campo si usa nello stesso modo del campo sin_port della struttura sockaddr_in.
- Il port number e' scritto (memorizzato) nel **network byte order.** (2Byte)

IPv6 Address Data Structures (2/3)

```
struct in6_addr {
          uint8_t  s6_addr[16];  // 128-bit IPv6 address (N.B.O.)
};
struct sockaddr_in6 {
  sa_family_t     sin6_family;   //AF_INET6
  in_port_t       sin6_port;     //transport layer port # (N.B.O.)
  uint32_t        sin6_flowinfo; //IPv6 flow information (N.B.O.)
  struct in6_addr sin6_addr;    // IPv6 address
  uint32_t        sin6_scope_id; //set of interfaces for a scope
}
```

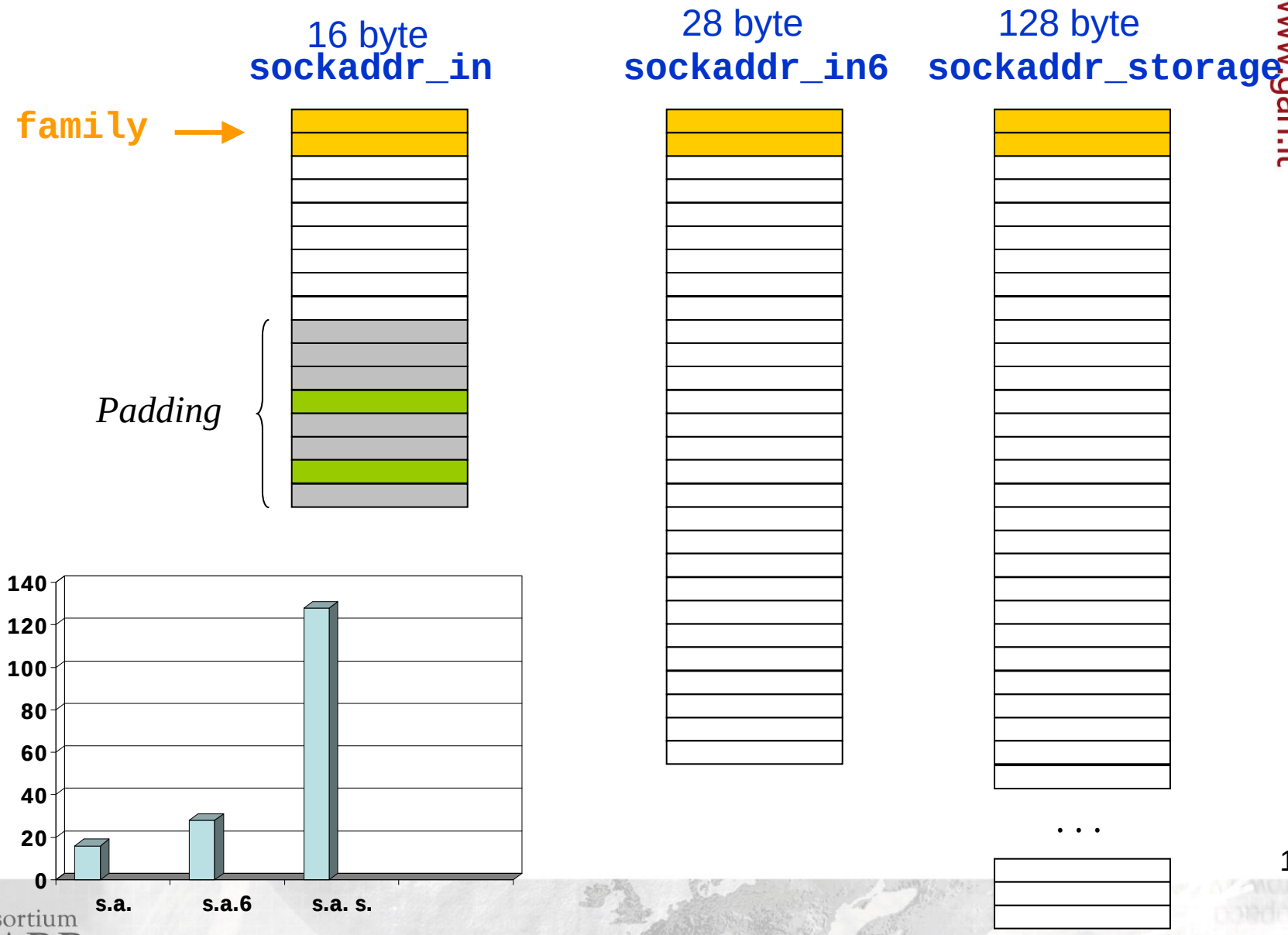

- **sin6_flowinfo** e' un campo a 32-bit field che contiene informazioni sul flusso.
- Il modo esatto in cui questo campo viene mappato su o da un pacchetto non e' attualmente definito.
- Fino a quando non verra' chiarito, le applicazioni dovrebbero settarlo a zero quando costruiscono una struttura **sockaddr_in6**, ed ignorarlo gestendo le **sockaddr_in**6 fornite dal sistema. (4Byte)
- **sin6_addr** e' una singola struttura **in6_addr**. **Questo campo contiene l'indirizzo IPv6 a 128-bit (uno solo).** L'indirizzo e' memorizzato nel **network byte order**. (16Byte)

IPv6 Address Data Structures (3/3)

```
struct in6_addr {
          uint8_t  s6_addr[16];  // 128-bit IPv6 address (N.B.O.)
};
struct sockaddr_in6 {
  sa_family_t     sin6_family;   //AF_INET6
  in_port_t       sin6_port;     //transport layer port # (N.B.O.)
  uint32_t        sin6_flowinfo; //IPv6 flow information (N.B.O.)
  struct in6_addr sin6_addr;    // IPv6 address
  uint32_t        sin6_scope_id; //set of interfaces for a scope
}
```


- **sin6_scope_id** e' un intero a 32 bit che identifica un insieme di interfacce come appropriate per lo scope dell'indirizzo dentro al campo **sin6_addr**.
- Il mapping di sin6_scope_id ad un interfaccia o un gruppo di interfacce e' lasciato all'implementazione ed a specifiche future al riguardo degli *scoped addresses*. (4Byte)
- **RFC 3493** non definisce l'uso del campo **sin6_scope_id.**
- Si intendeva specificarlo in seguito, ma fino ad ora non e' successo.

IPv4 / IPv6: Sockaddr_storage



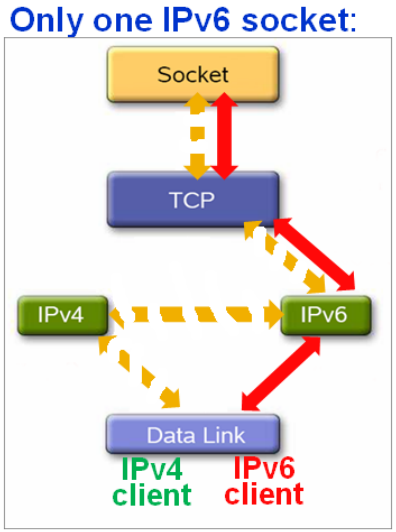
Socket Options

Per IPv6 sono state definite un certo numero di nuove socket options:

```
IPV6_UNICAST_HOPS
IPV6_MULTICAST_IF
IPV6_MULTICAST_HOPS
IPV6_MULTICAST_LOOP
IPV6_JOIN_GROUP
IPV6_LEAVE_GROUP
IPV6_V6ONLY
```

- Tutte queste nuove opzioni sono al livello di IPPROTO_IPV6
  - Che specifica il codice nel sistema per interpretatarle
- La dichiarazione di IPPROTO_IPV6 si ottiene includendo l'header <netinet/in.h>.

IPV6_V6ONLY



• **I socket appartenenti alla famiglia AF_INET6** si possono utilizzare sia per la comunicazione IPv4 che per quella IPv6.

- Il socket puo' anche essere utilizzato per inviare e ricevere solo pacchetti IPv6 utilizzando IPV6_V6ONLY.

- Per default e' spenta.

```
int on = 1;

if(setsockopt(s,IPPROTO_IPV6,IPV6_V6ONLY,(char *)&on,sizeof(on))==-1)
      perror("setsockopt IPV6_V6ONLY");
    else
      printf("IPV6_V6ONLY set\n");
```

Un esempio di uso di IPV6_V6ONLY e' per consentire di avere due versioni dello stesso processo server in esecuzione sulla stessa porta, una che gestisce il traffico IPv4 ed una che gestisce quello IPv6 (separando gli stack).

Esempio con IPV6_V6ONLY

```
struct sockaddr_in6 sin6, sin6_accept;
socklen_t sin6_len; int s0, s;  int on, off; char hbuf[NI_MAXHOST];

memset(&sin6,0,sizeof(sin6));
sin6.sin6_family=AF_INET6; sin6.sin6_len=sizeof(sin6);
sin6.sin6_port=htons(5001);

s0=socket(AF_INET6,SOCK_STREAM,IPPROTO_TCP);
on=1; setsockopt=(s0,SOL_SOCKET, SO_REUSEADDR, &on,sizeof(on));

#ifdef USE_IPV6_V6ONLY
  on=1;
  setsockopt(s0,IPPROTO_IPV6, IPV6_V6ONLY,&on,sizeof(on));
#else
  off=0;
  setsockopt(s0,IPPROTO_IPV6, IPV6_V6ONLY,&off,sizeof(off));
#endif

bind(s0,(const struct sockaddr *)&sin6, sizeof(sin6));
listen(s0,1);
while(1){
   sin6_len=sizeof(sin6_accept);
   s=accept(s0,(struct sockaddr *)&sin6_accept, &sin6_len);
   getnameinfo((struct sockaddr *)&sin6_accept, sin6_len, hbuf,
       sizeof(hbuf), NULL, 0, NI_NUMERICHOST);
   printf("accept a connection from %s\n",hbuf);
   close(s);
}
```

Eseguiamo il codice dell'esempio

Senza utilizzare **USE_IPV6_V6ONLY**:

```
telnet ::1 5001
```

```
Accept a connection from ::1
```

```
telnet 127.0.0.1 5001
```

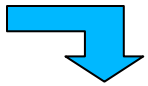
```
Accept a connection from ::ffff:127.0.0.1
```

Usando **USE_IPV6_V6ONLY**

```
telnet ::1 5001
```

```
Accept a connection from ::1
```

```
telnet 127.0.0.1 5001
```



```
Trying 127.0.0.1 …
telnet: connection to address 127.0.0.1: Connection refused
```

Riassunto su IPV6_V6ONLY

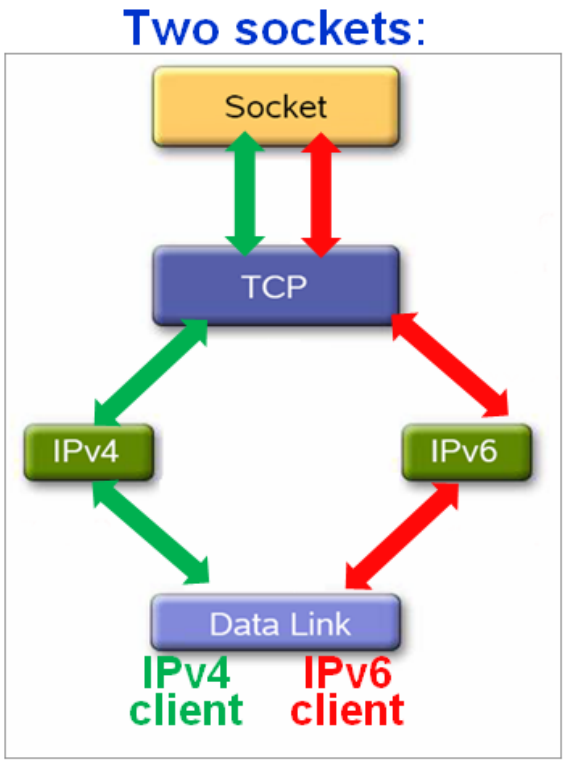
- 1) Se IPV6_V6ONLY = 1
  - a) Il socket accetta solamente connessioni IPv6
  - b) Si puo' creare un altro socket, IPv4, sulla stessa porta
- 2) Se IPV6_V6ONLY = 0
  - a)Il socket accetta entrambe, connessioni IPv4 ed IPv6
  - b)Non si puo' creare un altro socket IPv4 sulla stessa porta (darebbe come errore EADDRINUSE)
- Se non si setta a 0 o 1 IPV6_V6ONLY in setsockopt nel codice, il valore utilizzato e' quello scritto su

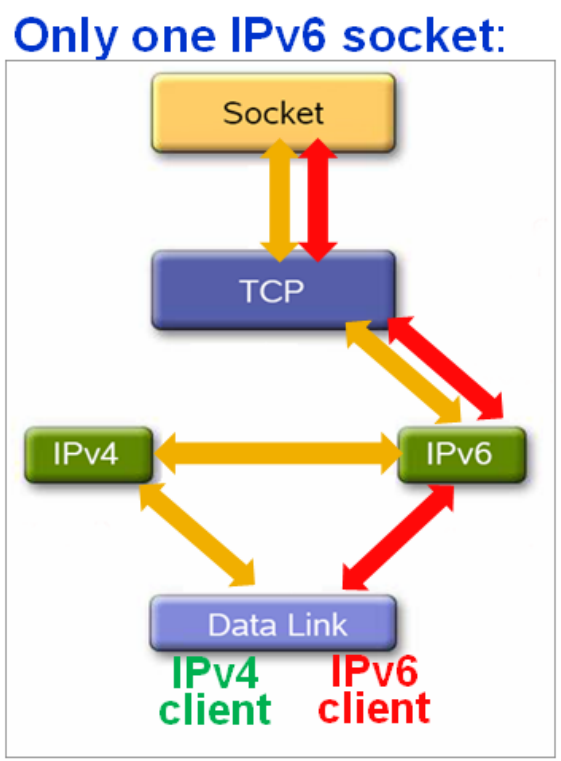
  /proc/sys/net/ipv6/bindv6only

  (che in linea di principio a-priori e' impredicibile per una applicazione) – percio' si rischia 1a o 2b

Un socket server: “Due sockets” e “Un solo socket IPv6”

Nelle prossime slides mostriamo esempi implementativi degli scenarii “Due Sockets Separati” o “Un solo socket (IPv6)”

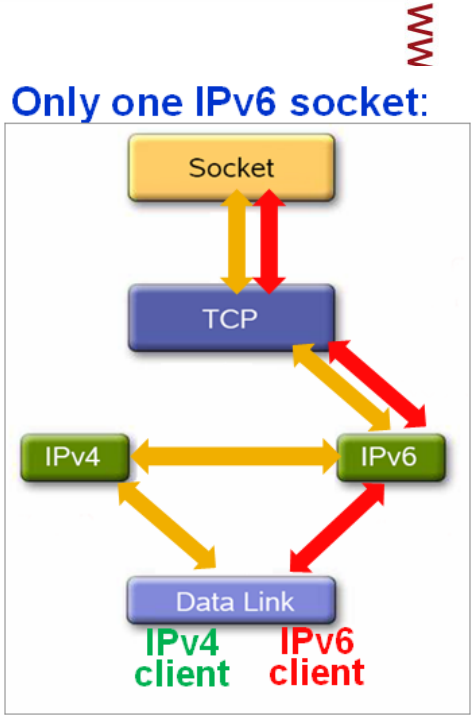




Socket Server: Un solo socket (IPv6)

```
int ServSock, csock;
struct sockaddr addr, from;
...
ServSock = socket(AF_INET6, SOCK_STREAM,PF_INET6);

bind(ServSock, &addr, sizeof(addr));
do {
        csock = accept(ServSocket, &from,sizeof(from));
        doClientStuff(csock);
} while (!finished);
```

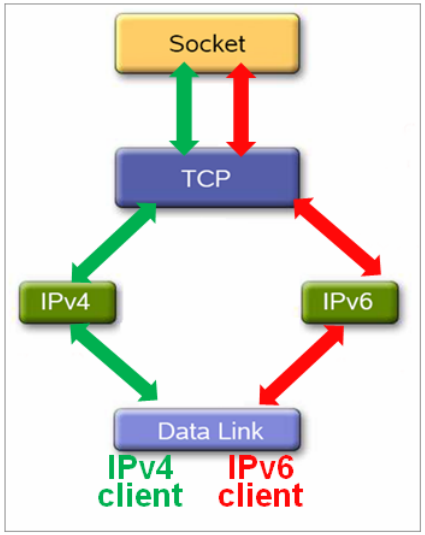



```
$ netstat -nap |grep 5002
tcp6   0      0 :::5002         :::*         LISTEN     3720/a.out
```

Socket Server: Due socket (1/3)

```
...
ServSock[0] = socket(AF_INET6, SOCK_STREAM, PF_INET6);
ServSock[1] = socket(AF_INET, SOCK_STREAM, PF_INET);
...
setsockopt(s[0], IPPROTO_IPV6, IPV6_V6ONLY, &on, sizeof(on))
...
bind(ServSock[0], AI0->ai_addr, AI0->ai_addrlen);
bind(ServSock[1], AI1->ai_addr, AI1->ai_addrlen);
...
select(2, &SockSet, 0, 0, 0);
if (FD_ISSET(ServSocket[0], &SockSet)) {
    // IPv6 connection
    csock = accept(ServSocket[0], (LPSOCKADDR)&From,FromLen);
    ...
}
if (FD_ISSET(ServSocket[1], &SockSet)) {
    // IPv4 connection
    csock = accept(ServSocket[1], (LPSOCKADDR)&From, FromLen);
    ...
}
```

Two sockets:




```
$ netstat -nap |grep 5002

tcp    0      0 0.0.0.0:5002    0.0.0.0:*    LISTEN      3720/a.out
tcp6   0      0 :::5002         :::*         LISTEN      3720/a.out
```

Socket Server: Due socket (2/3)

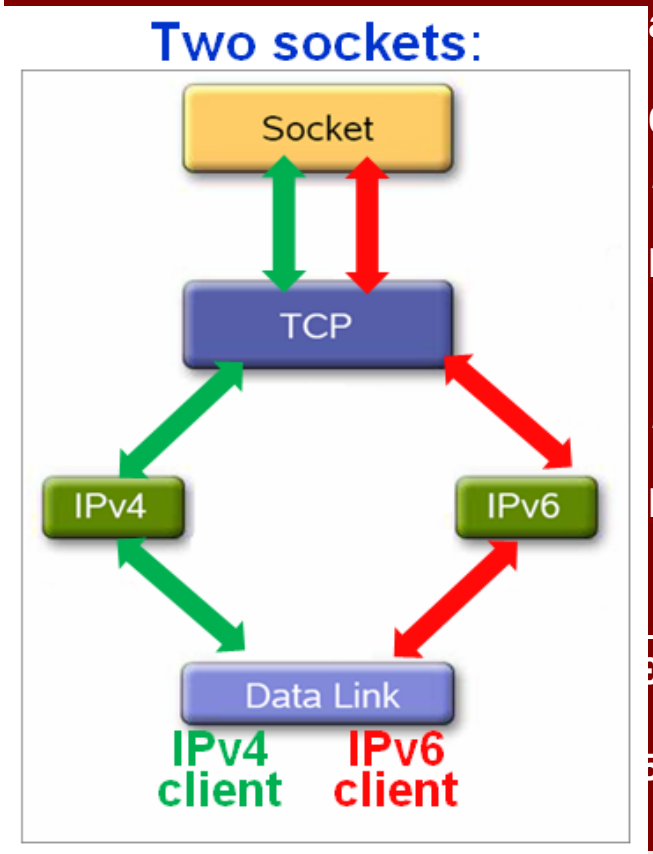

```
...
ServSock[0] = socket(AF_INET6, SOCK_STREAM, PF_INET6);
ServSock[1] = socket(AF_INET, SOCK_STREAM, PF_INET);
...
setsockopt(s[0], IPPROTO_IPV6, IPV6_V6ONLY, &on, sizeof(on));
...
bind(ServSock[0], AI0->ai_addr, AI0->ai_addrlen);
addr, AI1->ai_addrlen);

0);
, &SockSet)) {

ket[0], (LPSOCKADDR)&Fro

, &SockSet)) {

ket[1], (LPSOCKADDR)&Fro
```



L'opzione IPV6_V6ONLY consente a due versioni dello stesso processo server di ascoltare sulla stessa porta: una fornisce il servizio per IPv4 e l'altra per IPv6

```
02

5002    0.0.0.0:*    LISTEN      3720/a.out
        :::*         LISTEN      3720/a.out
```

Socket Server: Due socket (3/3)

Output del codice dell'esempio precedente:

CLIENT		SERVER
`$ telnet ::1 5002`	⇨	`IPv6 connection` `accept a connection from ::1`
`$ telnet 127.0.0.1 5002`	⇨	`IPv4 connection` `accept a connection from 127.0.0.1`

I due socket sono in ascolto sul server:

```
$ netstat -nap |grep 5002

tcp    0      0 0.0.0.0:5002    0.0.0.0:*    LISTEN     3720/a.out
tcp6   0      0 :::5002         :::*         LISTEN     3720/a.out
```

Riassunto sugli output dei server:

Un solo socket IPv6:

CLIENT **SERVER**

```
telnet 127.0.0.1 5001
```

```
Accept a connection from ::ffff:127.0.0.1
```

```
$ netstat -nap |grep 5001
tcp6   0      0 :::5001         :::*         LISTEN     3735/a.out
```

Due socket:

CLIENT **SERVER**

```
$ telnet 127.0.0.1 5002
```

```
IPv4 connection
accept a connection from 127.0.0.1
```

```
$ netstat -nap |grep 5002
tcp    0      0 0.0.0.0:5002    0.0.0.0:*    LISTEN     3720/a.out
tcp6   0      0 :::5002         :::*         LISTEN     3720/a.out
```

Funzioni per la Address Conversion


```
#include <netinet/in.h>

unsigneg long int htonl (unsigned long int hostlong)
unsigned short int htons (unsigned short int hostshort)
unsigned long int ntohl (unsigned long int netlong)
unsigned short int ntohs (unsigned short int netshort)
```

Le vecchie funzioni per la conversione degli indirizzi (che funzionano solo con IPv4) sono state sostituite da nuove, compatibili IPv6:


```
#include <arpa/inet.h>
int         inet_pton(int family, const char *src, void *dst);
const char *inet_ntop(int family, const void *src, char *dst,
size_t cnt);
```

La programmazione ***Network Transparent***

- Sono state definite nuove funzioni di rete per il supporto di entrambi i protocolli (IPv4 ed IPv6)
- E' stato introdotto un nuovo modo di programmare e di gestire i socket: *la programmazione* ***Network Transparent***
- I programmatori devono scrivere il loro codice senza assumere a-priori una specifica versione del protocollo IP (IPv4 o IPv6)
- In questo nuovo approccio alla programmazione di rete sono state definite:
  - **getaddrinfo()**
  - **getnameinfo()**
- Per la programmazione Network Transparent e' fondamentale fare attenzione a:
  - **In generale, per identificare i nodi, utilizzare i nomi e non gli indirizzi numerici**
  - **Non utilizzare mai indirizzi numerici hard-coded**
  - **Utilizzare *getaddrinfo* e *getnameinfo***

getaddrinfo()

Name to Address Translation Function: **getaddrinfo()**

gethostbyname() [per IPv4] e ***gethostnyname2()*** [originariamente creata per IPv6] **sono state deprecate** nell' RFC 2553 e rimpiazzate dalla funzione **getaddrinfo()**.

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name)
```

DEPRECATED

```
#include <netdb.h>
#include <sys/socket.h>
struct hostent *gethostbyname2(const char *name, int af);
```

DEPRECATED

getaddrinfo() accetta in ingresso il nome di un tipo di servizio (per es. "http") o un numero di porta (per es. "80") ed il FQDN e restituisce una lista di indirizzi ed i corrispondenti numeri di porta.

getaddrinfo e' molto flessibile ed ha molteplici modi di funzionamento. **Restituisce una linked list allocata dinamicamente di strutture di tipo addrinfo** che contiene informazione utile (per esempio una struttura sockaddr pronta per l'uso..).

```
#include <netdb.h>
#include <sys/socket.h>
int getaddrinfo(const char *nodename, const char *servname,
        const struct addrinfo *hints, struct addrinfo **res);
```

Nodename and Service Name Translation


```
int getaddrinfo(const char *nodename, const char *servname,
        const struct addrinfo *hints, struct addrinfo **res);
```

int getaddrinfo(...)

Function returns:
0 for success
not 0 for error
(see *gai_strerror*)

const char *nodename

Host name or
Address string

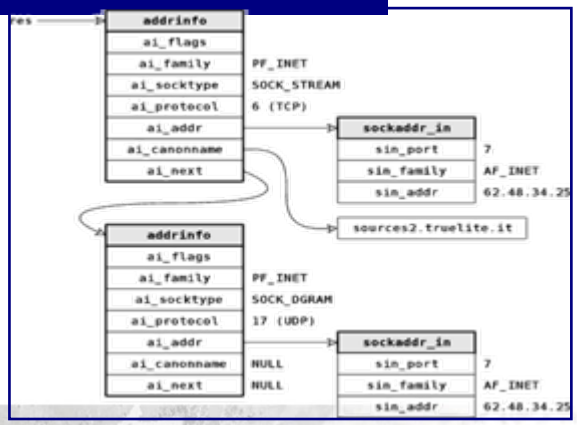
const char *servname

Servicename or
decimal port
("http" or 80)

const struct addrinfo *hints

Options
Es. nodename is
a numeric host addressing

struct addrinfo **res



getaddrinfo: parametri in input

Il chiamante puo' settare solo questi campi della struttura ***hints***:

```
struct addrinfo {
  int     ai_flags;     // AI_PASSIVE, AI_CANONNAME, ..
  int      ai_family;    // AF_xxx
  int      ai_socktype;  // SOCK_xxx
  int      ai_protocol;  // 0 or IPPROTO_xxx for IPv4 and IPv6
  socklen_t  ai_addrlen;   // length of ai_addr
  char   *ai_canonname; // canonical name for nodename
  struct sockaddr  *ai_addr; // binary address
  struct addrinfo  *ai_next; // next structure in linked list
};
```

ai_family: la famiglia di protocolli da ritornare (es. AF_INET, AF_INET6, **AF_UNSPEC**).

Quando *ai_family* e' settata a AF_UNSPEC, il chiamante accetta qualsiasi famiglia di protocollo supportata dal sistema operativo.

ai_socktype: indica il tipo di socket desiderato: SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW. Se *ai_socktype* e' zero →qualsiasi tipo.

ai_protocol: denota il protocollo di trasporto desiderato, IPPROTO_UDP o IPPROTO_TCP. Se *ai_protocol* e' zero si accetta qualsiasi prot.di trasporto.

Getaddrinfo - parametri in input: ai_flag (1/3)


```
struct addrinfo {
  int      ai_flags;     // AI_PASSIVE, AI_CANONNAME, ..
[…]
};
```

ai_flags si setta a zero o al OR bit-inclusivo di uno o piu' di questi valori:

AI_PASSIVE

Si richiedono indirizzi adatti ad accettare **connessioni entranti**. Con questa opzione in genere *nodename* e' NULL, ed il campo address del membro ai_addr e' riempito con l'indirizzo any (ovvero INADDR_ANY per IPv4 o IN6ADDR_ANY_INIT per IPv6).

AI_CANONNAME

La funzione prova a determinare il nome canonico corrispondente al node name (il primo elemento della lista ritornata ha il campo ai_canonname riempito con nome ufficiale del nodo)

getaddrinfo - parametri in input: ai_flag (2/3)

```
struct addrinfo {
  int      ai_flags;     // AI_PASSIVE, AI_CANONNAME, ..
[…]
};
```


AI_NUMERICHOST

Specifica che node name una strina numerica di host address.

Questa stringa inibisce ogni funzionalita' di tipo name-resolving (per esempio il DNS)

AI_NUMERICSERV

Specifica che servname e' una stringa numerica, la porta.

Questa flag inibisce ogni funzionalita' di tipo name resolution service (per es. NIS+)

AI_V4MAPPED

Se non ti trovano indirizzi IPv6, vengono allora restituiti indirizzi IPv6 di tipo IPv4-mapped corrispondenti agli indirizzi IPv4 che matchano nodename. Questa flag si puo' usare solo quando *ai_family* e' *AF_INET6* nella struttura *hints*.

getaddrinfo parametri in input: ai_flag (3/3)


```
struct addrinfo {
  int      ai_flags;     // AI_PASSIVE, AI_CANONNAME, ..
[…]
};
```

AI_ALL

Se si setta questa flag insieme ad AI_V4MAPPED quando si fa il look up di un indirizzo IPv6, la funzione restituisce tutti gli indirizzi IPv6 e tutti gli indirizzi IPv4 (mappati nel formato IPv4-mapped di IPv6)

AI_ADDRCONFIG

Saranno ritornati solo indirizzi la cui famiglia e' supportata dal sistema: indirizzi IPv4 verranno restituiti se c'e' un indirizzo IPv4 sul local system, e analogamente verranno ritornati indirizzi IPv6 se c'e' un indirizzo IPv6 sul sistema locale. (non si include in questo discorso l'interfaccia di loopback)

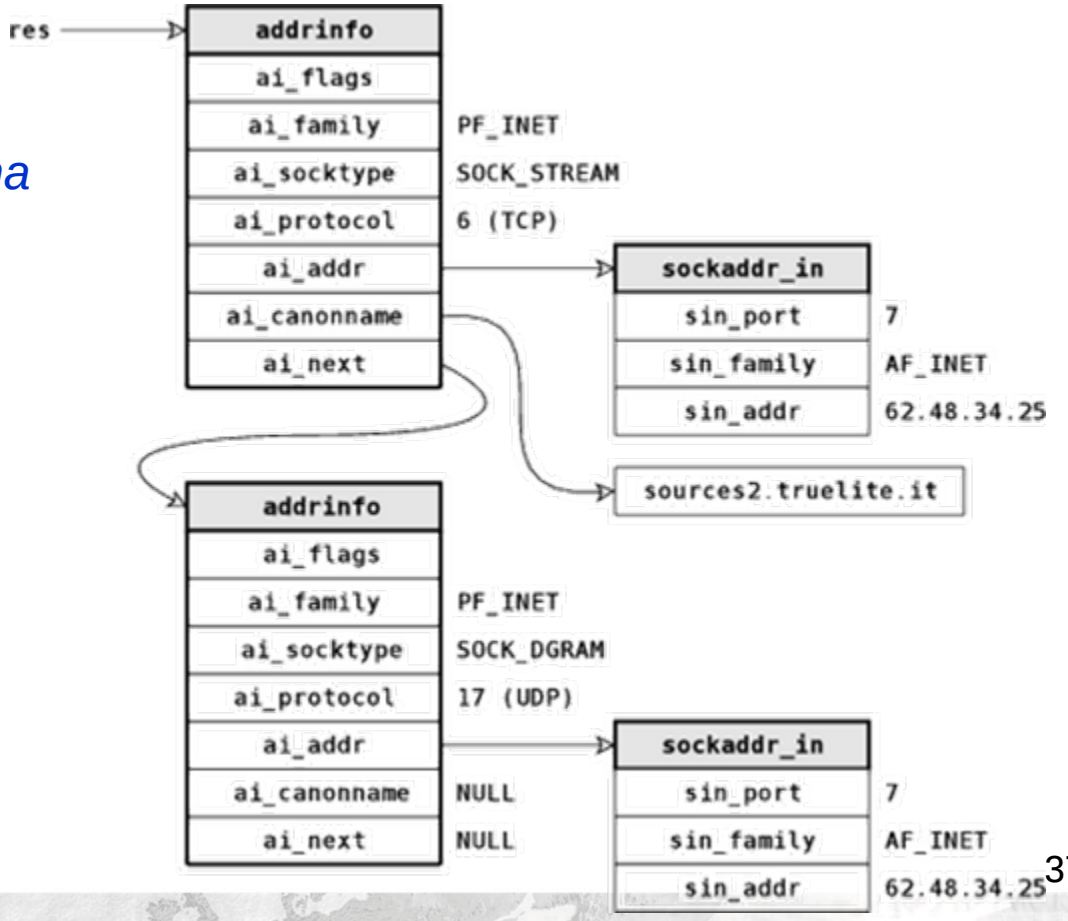
.

L' output di *getaddrinfo* (1/2)

Se getaddrinfo ritorna 0 (successo), l'argomento ***res*** viene riempito con un puntatore ad **una linked list di strutture *addrinfo*** (linkate attraverso ***ai_next_pointer***)

In caso di indirizzi multipli associati ad un hostname, una struttura e' restituita per ogni indirizzo (utilizzabile con hint.ai_family, se specificato)

Una struttura e' anche restituita per ogni tipo di socket (preso da hint.ai_socktype)



L' output di *getaddrinfo* (2/2)


```
struct addrinfo {
  int     ai_flags;     /* AI_PASSIVE, AI_CANONNAME, .. */
  int     ai_family;    /* AF_xxx */
  int     ai_socktype;  /* SOCK_xxx */
  int     ai_protocol;  /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
  socklen_t  ai_addrlen;   /* length of ai_addr */
  char   *ai_canonname; /* canonical name for nodename */
  struct sockaddr  *ai_addr; /* binary address */
  struct addrinfo  *ai_next; /* next structure in linked list */
};
```

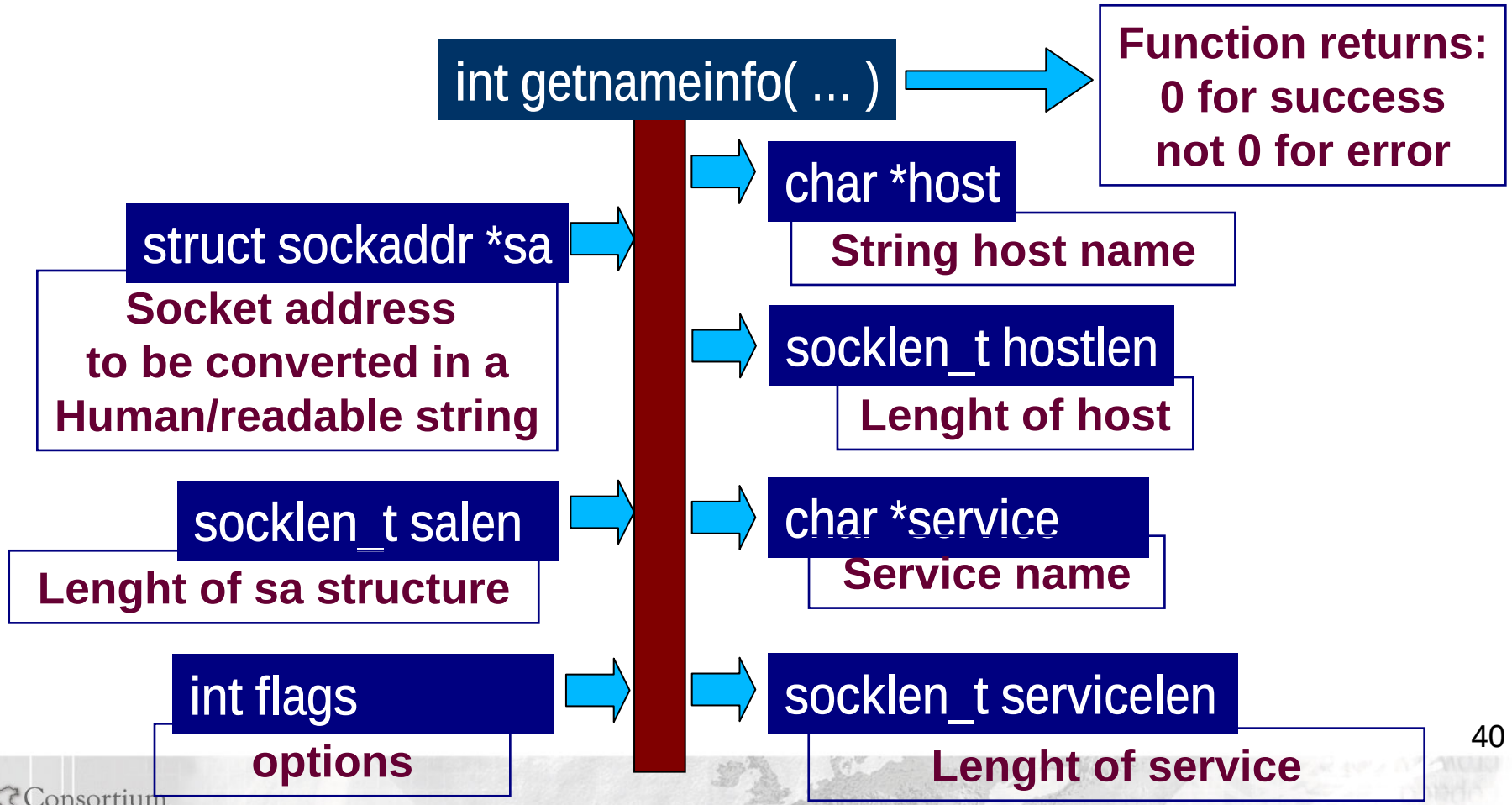
L'informazione restituita nelle addrinfo data structures e' pronta per le chiamate socket, e pronta per essere usata nelle funzioni di *connect, sendto* (per clients) e *bind* (per servers)

ai_addr e' un pointer ad una socket address structure.
ai_addrlen e' la lunghezza di questa socket address structure.
ai_canonname della prima struttura addrinfo restituita punta al canonical name del nodo (se AI_CANONNAME e' settata nella struttura *hints*)

getnameinfo()

Nodename e Service Name Translation

```
int getnameinfo (const struct sockaddr *sa,
  socklen_t salen,char *host, socklen_t hostlen,
  char *service, socklen_t servicelen,  int flags);
```



getnameinfo - input: *flags*


```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo (const struct sockaddr *sa, socklen_t salen,
                 char *host, socklen_t hostlen,
                 char *service, socklen_t servicelen,
                   int flags);
```

flags modifica il comportamento default della funzione.

Per default, il fully-qualified domain name (FQDN) dell' host viene restituito, ma :

- Se il flag bit NI_NOFQDN e' settato, solo la porzione node name del FQDN viene restituita per hosts locali
- Se la flag bit NI_NUMERICHOST e' settata, la forma numerica dell'indirizo dell'host viene restituita – anziche' il suo nome.
- […]

Due esempi di uso di getnameinfo

- Il primo illustra l'uso dei risultati di ***getaddrinfo*()** per le successive chiamate a socket() e a connect()

- Il secondo esempio apre passivamente socket in ascolto di connessioni HTTP in arrivo

getnameinfo esempio 1


```
struct addrinfo hints,*res,*res0; int error; int s;

memset(&hints,0,sizeof(hints));
hints.ai_family=AF_UNSPEC;
hints.ai_socktype=SOCK_STREAM;
error=getaddrinfo("www.kame.net","http",&hints,&res0);
[…]

s=-1;
for(res=res0; res; res=res->ai_next){
   s=socket(res->ai_family, res->ai_socktype,res->ai_protocol);
   if(s<0) continue;
   if(connect(s,res->ai_addr,res->ai_addrlen)<0){
        close(s); s=-1; continue;}
        break; // we got one!
   }
   if(s<0){fprintf(stderr,"No addresses are reachable");exit(1);}
   freeaddrinfo(res0);
}
```

getnameinfo esempio 2

```
    struct addrinfo hints, *res, *res0;
    int error; int s[MAXSOCK]; int nsock; const char *cause=NULL;
memset (&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.aisocktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
error=getaddrinfo(NULL,"http", &hints, &res0);
 nsock=0;
for(res=res0; res && nsock<MAXSOCK; res=res->ai_next)
{
 s[nsock]=socket(res->ai_family, res->ai_socktype, res->ai_protocol);
 if(s[nsock]<0) continue;
#ifdef IPV6_V6ONLY
  if(res->ai_family == AF_INET6){int on=1;
  if(setsockopt(s[nsock],IPPROTO_IPV6,IPV6_V6ONLY,&on,sizeof(on)))
  {close(s[nsock]);continue;}}
#endif
 if(bind(s[nsock], res->ai_addr, res->ai_addrlen)<0)
  {close(s[nsock]);continue;}
  if(listen(s[nsock],SOMAXCONN)<0){close(s[nsock]);continue;}
       nsock++;
}
if(nsock==0){ /*no listening socket is available*/}
freeaddrinfo(res0);
}
```

Introduzione alla programmazione IPv6 in Perl

Perl ed IPv6

1. Un insieme di funzioni IPv6 per Perl viene fornito dal modulo **Socket6**
2. Come il modulo Socket di core per IPv4, fornisce un set di funzione C-style per aprire e gestire Sockets in IPv6
3. La struttura generale del modulo e le address data structures sono simili a quelle del C.
4. Gli sviluppatori dovrebbero tenere in cosiderazione quanto detto per il C e la programmazione IPv6.
5. Il modulo e' disponibile sul web site di CPAN. Per lavorare in maniera corretta, il modulo deve essere incluso nel codice assieme al modulo Socket di core.

```
use Socket
Use Socket6
```

Lista delle funzioni (1/6)


```
BINARY_ADDRESS = inet_pton (FAMILY, TEXT_ADDRESS)
```

Questa funzione converte indirizzi stringa in formato IPv4/IPv6 in formato binario

L'argomento FAMILY specifica il tipo di indirizzi (AF_INET o AF_INET6)

```
TEXT_ADDRESS = inet_ntop (FAMILY, BINARY_ADDRESS)
```

Questa funzione converte un indirizzo in formato binario in uno formato stringa.

FAMILY specifica il tipo di indirizzi (AF_INET o AF_INET6)

Esempio:

```
$a=inet_ntop(AF_INET6,inet_pton(AF_INET6,"::1"));
print $a; //print ::1
```

Lista delle funzioni (2/6)


```
STRUCT_ADDR = pack_sockaddr_in6 (PORT, ADDRESS)
```

Questa funzione ritorna una struttura sockaddr_in6, con gli argomenti PORT (porta) ed ADDRESS (indirizzo) nei campi corretti. L'argomento ADDRESS e' una struttura 16-byte (come ritornato da inet_pton). I rimanenti campi della struttura non vengono settati.

```
(PORT,STRUCT_ADDR) = unpack_sockaddr_in6 (ADDR)
```

Questa funzione spacchetta una struttura sockaddr_in6 in un array di 2 elementi:

Il primo e' il numero di porta

Il secondo e' l'indirizzo incluso nella struttura

esempio

```
$lh6=inet_pton(AF_INET6,"::1");
$p_saddr6=pack_sockaddr_in6(80,$lh6);
($port,$host) = unpack_sockaddr_in6($p_saddr6);
print inet_ntop(AF_INET6,$host); //print ::1
print $port; //print 80
```

Lista delle funzioni (3/6)


```
pack_sockaddr_in6_all (PORT, FLOWINFO, ADDRESS, SCOPEID)
```

Questa funzione ritorna una struttura sockaddr_in6, riempiendola con i 4 argomenti specificati in input.

```
unpack_sockaddr_in6_all (NAME)
```

Questa funziona spacchetta una struttura sockaddr_in6 in un array di 4 elementi

- Numero di porta
- Flow information
- Indirizzo IPv6 (16-byte format)
- Lo *scope* dell'indirizzo

Lista delle funzioni (4/6)


```
getaddrinfo(NODENAME,SERVICENAME,[FAMILY,SOCKTYPE,PROTOCOL,FLAGS])
```

Questa funzione **converte i nomi dei nodi in indirizzi** ed **i nomi dei servizi in numero di porta**.

Almeno uno tra NODENAME e SERVICENAME deve avere un valore vero.

Se il lookup va a buon fine, questa funzione restituisce **un array di blocchi di informazione.**

Ogni blocco di informazione ha 5 elementi: **address family, socket type, protocol, address and canonical name** se specificato.
Gli argomenti in parentesi quadrata [] sono opzionali

```
getnameinfo (NAME, [FLAGS])
```

Questa funzione restituisce un nome di nodo o di servizio. L'attributo opzionale FLAGS determina che tipo di look up viene effettuato.

Esempio 1

```
use Socket;
use Socket6;
@res = getaddrinfo('hishost.com', 'daytime', AF_UNSPEC,SOCK_STREAM);
 $family = -1;
 while (scalar(@res) >= 5) {
  ($family, $socktype, $proto, $saddr, $canonname, @res)=@res;
  ($host, $port) =getnameinfo($saddr,NI_NUMERICHOST|NI_NUMERICSERV);
   print STDERR "Trying to connect to $host port $port...\n";

   socket(Socket_Handle, $family, $socktype, $proto) || next;
   connect(Socket_Handle, $saddr) && last;
   close(Socket_Handle);
   $family = -1;
 }

 if ($family != -1) {
   print STDERR "connected to $host port port $port\n";
 } else {
   die "connect attempt failed\n";
 }
```

Esempio 2

```
use Socket; use Socket6;      use IO::Handle;      $family = -1;

@res = getaddrinfo('www.kame.net', 'http', AF_UNSPEC, SOCK_STREAM);
while (scalar(@res) >= 5) {
  ($family, $socktype, $proto, $saddr, $canonname, @res) = @res;
  ($host,$port) = getnameinfo($saddr,NI_NUMERICHOST|NI_NUMERICSERV);
  print STDERR "Trying to connect to $host port $port...\n";

  socket(Socket_Handle, $family, $socktype, $proto) || next;

  connect(Socket_Handle, $saddr) && last;
  close(Socket_Handle); $family = -1;
}

if ($family != -1) { Socket_Handle->autoflush();
    print Socket_Handle "GET\n";
    print STDERR "connected to $host port port $port\n";
    while($str=<Socket_Handle>){print STDERR $str;}
}else {die "connect attempt failed\n";}
```

Output dell'esempio 2


```
Trying to connect to 2001:200:0:8002:203:47ff:fea5:3085 port 80 …
connected to 2001:200:0:8002:203:47ff:fea5:3085 port 80

[…]<title>The KAME project</title>[…] The KAME project […]
<img src="/img/kame-anime-small.gif" alt="Dancing kame" /> […]
```

Output dell'esempio 2

```
(…)=getnameinfo($saddr,NI_NUMERICHOST|NI_NUMERICSERV);
print STDERR "Trying to connect to $host port $port…";
```

OUTPUT:

```
Trying to connect to 2001:200:0:8002:203:47ff:fea5:3085 port 80 …
```

```
(…)=getnameinfo($saddr,0);
print STDERR "Trying to connect to $host port $port…";
```

OUTPUT:

```
Trying to connect to orange.kame.net port www …
```

Lista delle funzioni (5/6)

gethostbyname2 (HOSTNAME, FAMILY)

Questa funzione e' l'implementazione multi-protocollo di gethostbyname

L'attributo FAMILY serve a selezionare la famiglia di indirizzi

Questa funzione risolve nomi nodo in indirizzi.

gai_strerror (ERROR_NUMBER)

Quetsa funzione restituisce una stringa corrispondente al numero di errore che viene passato in ingresso.

in6addr_any

Questa funzione restituisce l'indirizzo wildcard a 16byte.

in6add_loopback

Questa funzione restituisce l'indirizzo loopback a 16 byte.

Lista delle funzioni (6/6)


```
getipnodebyname (HOST, [FAMILY, FLAGS])
```

Questa funzione prende in ingresso un nome di nodo o un stringa che rappresenti un indirizzo IP ed esegue il lookup su quel nome (o sul risultato della conversione della stringa).

Restituisce 5 elementi: canonical host name, address family, lunghezza in byte dell' indirizzo IP restituito, una reference alla lista di IP data structures ed una reference ad una lista di alias per quell'host.

E' stata **deprecata** nel RFC3493. **getnameinfo** deve essere usata al suo posto.

```
getipnodebyaddr (FAMILY, ADDRESS)
```

Questa funzione prende in ingresso una famiglia di indirizzi IP ed una struttura di indirizzo IP ed effettua il reverse lookup su quell'indirizzo.

E' stata **deprecata** dal RFC3493: **getaddrinfo** deve essere usata al suo posto.

Programmazione IPv6 in Python in 2 slides: solo un paio di indicazioni

Python ed IPv6

- Python supporta IPv6 dalla v2.3 su Linux/Solaris
- Le funzioni IPv6 principali sono simili al C:
  - **getnameinfo, getaddrinfo**
- Inconvenienti di usare Python in IPv6:
  - La maggior parte del codice networking esistente richiede modifiche per diventare IPv6 compliant.
    - Tuttavia queste modifiche sono relativamente facili.
  - Non esiste (ancora) il simbolo IPv6_V6ONLY
    - Va definito a mano

Principali funzioni python IPv4/IPv6

Purpose of the function call	function
Get the list of addresses to listen on	getaddrinfo()
Create the server socket	**socket()**
Bind the server socket	**bind()**
Listen on the server socket	**listen()**
Choose if the IPv6 socket should accept IPv4 connections or not	setsockopt(..., IPV6_V6ONLY, ...)
Accept a client connection	**accept()**

Purpose of the function call	function
Return the list of addresses to connect to the server host	getaddrinfo()
Create the client socket	**socket()**
Get a character string representing an IP address	**getnameinfo (..., NI_NUMERICHOST)**
Connect to the server	**connect()**

Introduzione alla programmazione IPv6 in Java

IPv6 e Java (1/3)

- Le API Java sono gia' compliant IPv4 / IPv6.
- Il supporto per IPv6 in Java e' disponibile dalle versioni
  - **1.4.0 su Solaris e Linux**
  - **1.5.0 su Windows XP e 2003 server**.
- Il supporto IPv6 in Java e' implicito e trasparente.
- Infatti non servono modifiche al codice sorgente (tantomeno al bytecode! ☺)
- Ogni applicazione Java e' gia' IPv6 enabled se:
  - **Non utilizza indirizzi hard-coded** (non ha riferimenti diretti ad indirizzi IPv4..)
  - **Tutte le informazioni sull'indirizzo e sul socket sono incapsulate nelle API di Networking di Java**
  - Settando proprieta' di sistema, tipo di indirizzo e/o di socket si possono settare delle preferenze
  - Se non usa funzioni non specifiche per la risoluzione degli indirizzi

IPv6 e Java (2/3)

- NOTA BENE: in Java

  L' indirizzo IPv4-mapped ha significato solo a livello implementativo dello stack a due protocolli e non viene mai restituito (ritornato)

  - Corrispondentemente, viene invece restituito l'indirizzo IPv4 pieno, standard
- Per nuove applicazioni, si possono utilizzare nuove classi e nuove API specifiche per IPv6

IPv6 e JAVA (3/3): vantaggi e svantaggi

- **<u>Vantaggio:</u> La maggior parte del codice esistenze e' gia' IPv6 compliant**
- **<u>Vantaggio:</u> Il codice Java di networking e' generalmente molto sintetico e breve, perche' le funzionalita' principali sono gestite internamente dentro l'ambiente Java in maniera trasparente**
- **<u>Svantaggio:</u> Il parametro**

  /proc/sys/net/ipv6/bindv6only deve essere settata 0 **sul sistema operativo dove vengono eseguiti I programi. Altrimenti:**
  - Un server (IPv6) non accettera' client IPv4
  - Un client (IPv6) non riesce a connetersi ad un server IPv4 (dira' che il Network e' unreachable)
    - E' un bug di Java riportato nel sul bug DB

Esempio di codice Java (server):

Notate che sono esattamente le stesse linee di codice per IPv4 ed IPv6

- Basato sulla classe **ServerSocket**

```
import java.io.*;
import java.net.*;

ServerSocket serverSock = null;
Socket cs = null;

try {
 serverSock = new ServerSocket(5000);
 cs = serverSock.accept();
 BufferedOutputStream b = new
             BufferedOutputStream(cs.getOutputStream());
 PrintStream os = new PrintStream(b,false);
 os.println(“hallo!”); os.println("Stop");

 cs.close();
 os.close();
}catch (Exception e) {[...]}
```

Esempio di codice Java (client):

Notate che sono esattamente le stesse linee di codice per IPv4 ed IPv6

- Basato sulla classe **Socket**

```
import java.io.*;
import java.net.*;

Socket s = null; DataInputStream is = null;

try {
  s = new Socket("localhost", 5000);
  is = new DataInputStream(s.getInputStream());
  String line;
  while( (line=is.readLine())!=null ) {
    System.out.println("received: " + line);
    if (line.equals("Stop")) break;
  }
  is.close();
  s.close();
}catch (IOException e) { […] }
```

La classe **InetAddress**

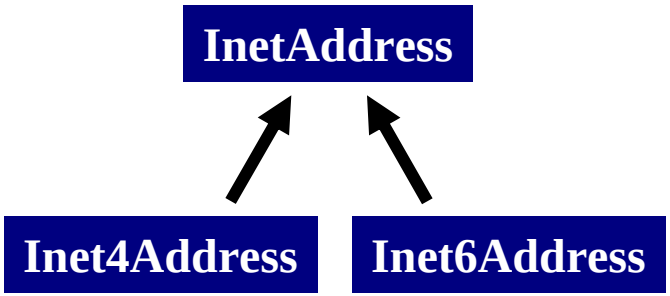
- Questa classe rappresenta un indirizzo IP. Fornisce:
  - Memorizzazione dell' indirizzo
  - Metodi per name-address translation
  - Metodi per address conversion
  - Metodi per address testing

Inet4Address ed Inet6Address

In J2SE 1.4, la classe InetAddress e' stata estesa per supportare indirizzi sia IPv4 che IPv6

Metodi utility sono stati aggiunti per verificare il tipo di indirizzo e lo scope

```
public final class Inet4Address extends InetAddress
public final class Inet6Address extends InetAddress
```



Inet4Address ed Inet6Address

- I due tipi di indirizzi, IPv4 ed IPv6, possono essere distinti dal Java class type Inet4Address e Inet6Address.
- Stati speficici e comportamenti V4 and V6 sono implementati in queste due sotto-classi.
- A causa della natura OO di Java, una applicazione normalmente ha bisogno di interagire solo con la classe madre InetAddress – e poi attraverso il polimorfismo ricevera' il trattamento corretto.
- Solo quando serve accedere a comportamenti specifici del protocollo, come per es. chiamare un metodo IPv6-only, o quando serve conoscere la class type dell'indirizzo IP, serve allora scendere al livello delle sotto-classi Inet4Address e Inet6Address

La classe InetAddress


```
public static InetAddress getLocalHost()
                                    throws UnknownHostException
```

Restituisce il local host

```
public static InetAddress getByName(String host)
                                    throws UnknownHostException
```

Determina l'indirizzo IP di un host a partire dal suo hostname

La clase InetAddress


```
public byte[] getAddress()
```

Restituisce l'indirizzo IP raw di questo oggetto InetAddress. Il risultato e' nel network byte order: il byte di ordine piu' alto ha l'indirizzo getAddress()[0]

```
InetAddress addr=InetAddress.getLocalHost();
byte[] b=addr.getAddress();
for(int i: b){System.out.print(i+" ");}
```

```
Output:
127 0 0 1
```

```
public static InetAddress getByAddress(byte[] addr)
                                        throws UnknownHostException
```

Restituisce un oggetto di classe InetAddress a partire dall'indirizzo raw.

Il risultato e' nel network byte order: il byte di ordine piu' alto ha l'indirizzo getAddress()[0]

Questo metodo non e' bloccante, ovvero non viene effettuato il reverse name service lookup

Array che contengono indirizzi IPv4 sono di 4 bytes, 16 bytes per un indirizzo IPv6

La classe InetAddress


```
public static InetAddress[] getAllByName(String host)
                                        throws UnknownHostException
```

Dato il nome di un host, restituisce un array dei suoi indirizzi IP, basandosi sul DNS configurato sul sistema.

```
for (InetAddress ia : InetAddress.getAllByName("www.kame.net")) {
            System.out.println(ia);
}
```

```
output:
  www.kame.net/203.178.141.194
  www.kame.net/2001:200:0:8002:203:47ff:fea5:3085
```

La classe InetAddress


```
public String getCanonicalHostName()
```

Ottiene il fully qualified domain name (FQDN) per un dato indirizzo IP.

E' un metodo best-effort: dipendentemente dalla configurazione del sistema, si potrebbe non ottenere l' FQDN

```
System.out.println(
   InetAddress.getByName("www.garr.it").getCanonicalHostName()
);
```

```
output:
  lx1.dir.garr.it
```

```
public String getHostAddress()
```

Restituisce la stringa con l'indirizzo IP in modo testuale.

```
addr = InetAddress.getByName("www.garr.it");
System.out.println(addr.getHostAddress());
```

```
output:
   193.206.158.2
```

La classe InetAddress


```
public String getHostName()
```

Restituisce l'hostname per un dato indirizzo IP.

Se questa istanza InetAddress e' stata creata con un hostname, verra' ricordato e restituito questo hostname iniziale.

Altrimenti si effettua un reverse DNS lookup ed il risultato verra' restituito, basandosi sul DNS definito nel sistema.

```
System.out.print(
      InetAddress.getByName("www.garr.it").getHostName()
);
```

```
output:
   www.garr.it
```

```
System.out.print(
    InetAddress.getByName("193.206.158.2").getHostName()
);
```

```
output:
   lx1.dir.garr.it
```

La classe InetAddress

```
public boolean isReachable(int timeout)
                                        throws IOException
```

```
public boolean isReachable(NetworkInterface netif,int ttl,int
timeout)
                                        throws IOException
```

Esegue un test per verificare la raggiungibilita' di un indirizzo.

Un'implementazione tipica usa ICMP ECHO REQUESTs , se si ha il permesso di farlo. Altrimenti prova a stabilire una connessione TCP sulla porta 7 (echo) dell'host di destinazione.

netif – L'interfaccia attraverso la quale fare il test, o null per un'interfaccia qualsiasi
ttl - maximum numbers of hops per provare - o 0 per il valore default
timeout – tempo in ms dopo il quale la call abortisce

Esempio di InetAddress


```
InetAddress ia=InetAddress.getByName("www.garr.it");
//or
InetAddress ia=InetAddress.getByName(“[::1]"); //or "::1"

String host_name = ia.getHostName();
System.out.println( host_name ); // ip6-localhost

String addr=ia.getHostAddress();
System.out.println(addr); //print IP ADDRESS
```

```
InetAddress[ ] alladr=ia.getAllByName("www.kame.net");
for(int i=0;i<alladr.length;i++) {
                  System.out.println(  alladr[i]  );  }
Output:
  www.kame.net/203.178.141.194
  www.kame.net/2001:200:0:8002:203:47ff:fea5:3085
```

Nuovi metodi

Alle classe **InetAddress** sono stati aggiunti nuovi metodi:

```
InetAddress.isAnyLocalAddress()
InetAddress.isLoopbackAddress()
InetAddress.isLinkLocalAddress()
InetAddress.isSiteLocalAddress()
InetAddress.isMCGlobal()
InetAddress.isMCNodeLocal()
InetAddress.isMCLinkLocal()
InetAddress.isMCSiteLocal()
InetAddress.isMCOrgLocal()
InetAddress.getCanonicalHostName()
InetAddress.getByAddr()
```

Inet6Address ha un metodo ulteriore rispetto a Inet4Address:

```
Inet6Address.isIPv4CompatibleAddress()
```

Esempio generale *(networkInt.java)*

```
Enumeration netInter = NetworkInterface.getNetworkInterfaces();
 while ( netInter.hasMoreElements() )
 {
   NetworkInterface ni = (NetworkInterface)netInter.nextElement();
   System.out.println( "Net. Int. : "+ ni.getDisplayName() );
   Enumeration addrs = ni.getInetAddresses();
     while ( addrs.hasMoreElements() )
     {
       Object o = addrs.nextElement();
       if ( o.getClass() == InetAddress.class ||
            o.getClass() == Inet4Address.class ||
            o.getClass() == Inet6Address.class )
       {
         InetAddress iaddr = (InetAddress) o;
         System.out.println( iaddr.getCanonicalHostName() );
         System.out.print("addr type: ");
         if(o.getClass() == Inet4Address.class) {…println("IPv4");}
         if(o.getClass() == Inet6Address.class){…println( "IPv6");}
         System.out.println( "IP: " + iaddr.getHostAddress() );
         System.out.println("Loopback? "+iaddr.isLoopbackAddress());
         System.out.println("SiteLocal?"+iaddr.isSiteLocalAddress());
         System.out.println("LinkLocal?"+iaddr.isLinkLocalAddress());
       }
     }
 }
```

Output dell'esempio generale

```
Net. Int. : eth0
------------------------
CanonicalHostName: fe80:0:0:0:212:79ff:fe67:683d%2
addr type: IPv6   IP: fe80:0:0:0:212:79ff:fe67:683d%2
Loopback? False  SiteLocal? False   LinkLocal? true
------------------------
CanonicalHostName: 2001:760:40ec:0:212:79ff:fe67:683d%2
addr type: IPv6   IP: 2001:760:40ec:0:212:79ff:fe67:683d%2
Loopback? False   SiteLocal? False   LinkLocal? false
------------------------
CanonicalHostName: pcgarr20.dir.garr.it
addr type: IPv4   IP: 193.206.158.140
Loopback? False   SiteLocal? False  LinkLocal? false

Net. Int. : lo
------------------------
CanonicalHostName: ip6-localhost
addr type: IPv6   IP: 0:0:0:0:0:0:0:1%1
Loopback? True   SiteLocal? False   LinkLocal? false
------------------------
CanonicalHostName: localhost
addr type: IPv4   IP: 127.0.0.1
Loopback? True   SiteLocal? False   LinkLocal? false
```

Networking Properties IPv6: preferIPv4Stack

`java.net.preferIPv4Stack (default: false)`

Se IPv6 e' disponibile sul sistema operativo, il socket nativo sottostante sara' un socket IPv6.

Questo consente alle applicazioni Java di connettersi a e accettare connessioni da, hosts sia IPv4 che IPv6

Se una appliacazione ha una preferenza per usare solo socket IPv4, allora puo' settare questa property a true.

La conseguenza e' che l'applicazione non sara' capace di comunicare con host IPv6.

java.net.preferIPv4Stack Esempio (1/3)


```
$ java networkInt

Net. Int. : eth0
-------------------------
CanonicalHostName: fe80:0:0:0:212:79ff:fe67:683d%2
IP: fe80:0:0:0:212:79ff:fe67:683d%2
-------------------------
CanonicalHostName: 2001:760:40ec:0:212:79ff:fe67:683d%2
IP: 2001:760:40ec:0:212:79ff:fe67:683d%2
-------------------------
CanonicalHostName: pcgarr20.dir.garr.it
IP: 193.206.158.140

Net. Int. : lo
-------------------------
CanonicalHostName: ip6-localhost
IP: 0:0:0:0:0:0:0:1%1
-------------------------
CanonicalHostName: localhost
IP: 127.0.0.1
```

java.net.preferIPv4Stack Esempio (2/3)


```
$ java -Djava.net.preferIPv4Stack=true networkInt

Net. Int. : eth0
-------------------------
CanonicalHostName: pcgarr20.dir.garr.it
IP: 193.206.158.140

Net. Int. : lo
-------------------------
CanonicalHostName: localhost
IP: 127.0.0.1
```

java.net.preferIPv4Stack Esempio (3/3)

Per configurare **java.net.preferIPv4Stack** e' possibile usare l'opzione **–D mentre si lancia l'applicazione**

```
$ java -Djava.net.preferIPv4Stack=true networkInt
```

… o configurare questa proprieta' direttamente nel codice sorgente:

```
System.setProperty("java.net.preferIPv4Stack","true");
```

```
Properties p = new Properties(System.getProperties());
p.setProperty("java.net.preferIPv6Addresses", "true");
System.setProperties(p);
```

IPv6 Networking Properties: preferIPv6Addresses

`java.net.preferIPv6Addresses (default: false)`

Se IPv6 e' disponibile sul sistema operativo, la preferenza default e' preferire un indirizzo IPv6 di tipo IPv4-mapped ad uno IPv6 semplice.

Questo per ragioni di backward compatibility – per esempio per applicazioni che dipendono dall'accesso ad un servizio IPv4-only, o per applicazioni che dipendono dalla rappresentazione %d.%d.%d.%d di un indirizzo IP.

Questa proprieta' puo' essere usata per cambiare la preferenza all'uso di indirizzi IPv6 IPv4-mapped per IPv4 rispetto all'uso di indirizzi IPv4.

Questo consente di testare ed installare applicazioni in ambienti dove l'appliacazione si deve connettere a servizi IPv6.

java.net.preferIPv6Addresses Esempio


```
//System.setProperty("java.net.preferIPv6Addresses",“false");
InetAddress ia=InetAddress.getByName("www.kame.net");
String ss=ia.getHostAddress();
System.out.println(ss); //print 203.178.141.194


System.setProperty("java.net.preferIPv6Addresses","true");
InetAddress ia=InetAddress.getByName("www.kame.net");
String ss=ia.getHostAddress();
System.out.println(ss);//print 2001:200:0:8002:203:47ff:fea5:3085
```

```
$java -Djava.net.preferIPv6Addresses=true -jar test.jar
2001:200:0:8002:203:47ff:fea5:3085

$java -Djava.net.preferIPv6Addresses=false -jar test.jar
203.178.141.194
$java -jar test.jar
203.178.141.194
```

Librerie di programmazione di alto livello
(High Level Libraries)

- Perche’ utilizzare librerie di alto livello
- Python
- Perl
- C/C++
- Altre tecniche

Perche' usare librerie di alto livello ?

- L' High-level code e' piu' corto
- L'High-level code e' piu' facile da leggere
- L'High-level code non e' condizionato (o lo e' poco) da possibili cambiamenti nelle API di basso livello

High-Level Python

- In Python ci sono High-level networking libraries, ma hanno svariati svantaggi
- Un esempio: la classe **ThreadingTCPServer**
  - Uso di base:

```
…
ThreadingTCPServer.address_family = socket.AF_INET6
server = ThreadingTCPServer(("", port), <handler>)
server.serve_forever()
```

  - Notiamo che:
    - Questa classe per default e' IPv4-only, percio' dobbiamo indicare esplicitamente la famiglia di indirizzi IPv6
    - Percio' il nostro codice non e' **address-family independent** (il codice fallisce se non c'e' IPv6 sul sistema operativo)
    - Questa classe non copre tutta la casistica: l'opzione IPv6_V6ONLY deve essere unset per consentire a client IPv4 di connettersi al server IPv6

High-Level Python

- Way out:
  - E' possibile creare una subclass di ThreadingTCPServer che risolve questi 3 problemi
  - Si guardi per es.il doc di EGEE SA2 "IPv6 programming with C/C++, Perl, Python and Java" su https://edms.cern.ch/document/971407

High-Level Perl

- Ci sono High-level networking libraries in Perl, ma la maggior parte non sono compliant IPv6.

- La piu' famosa – che e' IPv6 compliant e' **IO::Socket::INET6.**

High-Level C/C++

- La nuova libreria boost::asio (disponibile a partire da boost 1.35) si puo' usare in una IPv6 compliant way
  - Per i clients, guardare per esempio il codice esempio "Synchronous TCP daytime client" del tutorial boost::asio tutorial – che e' IPv6 compliant.
  - Per i server, l'unico accorgimento e' di evitare di usare il costruttore di tcp::acceptor che apre,binda e ascolta automaticamente
  - Infatti bisogna settare l'opzione socket IPV6_V6ONLY a 0; e lo si puo' fare solo dopo l'open e prima del bind.
  - Come settare questa opzione con boost: **`acceptor.set_option(ip::v6_only(false));`**
- Per ulteriori informazioni: https://edms.cern.ch/document/935729

Altre tecniche High Level

- Per un socket server con funzionalita TCP di base, e' possibile costruire un servizio basandosi su xinetd:
  - Xinetd gestisce il low-level networking code
  - Xinetd puo' gestire la IPv6 compliance di un servizio aggiungendo l'opzione **"flags=IPv6"** nel file **/etc/xinetd.d/<service_name>.**
  - In questo caso xinetd creera' un socket IPv6 che accettera' sia connessioni IPv4 che IPv6.